ONLINE AND REINFORCEMENT LEARNING
ASSIGNMENT 2

Soumya Mukherjee

CH24M571

A REPORT ON THE ALGORITHM USED AND ITS JUSTIFICATION

Introduction :

The question states the following conditions :

> There is a secret word chosen from a known dictionary.

> Each guess gives us feedback:

dogs → letters in the correct position.

cats → correct letters but in the wrong position.

The task: minimize the number of guesses (solve quickly) while also not wasting too much computation per guess.

Methodology :

The MinMax algorithm should be the best solution as it is the true worst case minimiser but its very slow per guess since its complexity is O(n^2) . It picks the guess that minimizes the size of the largest remaining candidate group after the feedback, so in the worst case we can eliminate as many possibilities as possible each turn . Since in our case N = 4300 s , the time required to process all records would be huge .

Simple Elimination i.e Heuristics on the other hand is very fast , since it filters based on the feedback . But when the candidate set is huge (4000+ words), eliminating only a few at a time is inefficient and it takes many turns .

Hence , what I have tried to implement is a hybrid approach where minmax is performed on the initial few words and based on a threshold we switch to simple greedy heuristics .

I am switching beyond a threshold because minimax is only worth the cost when the search space is massive. Once the candidate set is narrowed, a heuristic solver is nearly as effective, but far cheaper to run. This hybrid balances speed and accuracy.

> The early minimax step prevents us from wasting 5–6 turns exploring blindly.

> The later heuristic step prevents the solver from wasting CPU time doing huge minimax simulations on an already small set.

Discussing optimality

What I feel is to define what is optimal ..

If we think of having the minimal number of guesses , a true minmax would always be the best. This is because minimax explores all possible feedback partitions and chooses the guess that minimizes the worst-case branch size.This does not take into account the time complexity required to execute the code

A simple greedy frequency heuristic where we score each candidate by sum of letter frequencies in the current candidate set would be the absolute best in terms of time taken .

In my algorithm , there are tradeoffs too .

The fixed threshold to switch may not be perfect for all dictionary sizes. Sometimes, minimax would still be worth it for 3–4 steps if space is huge, but this switch cuts it off. Conversely, for small dictionaries, even 1 step of minimax may be wasted.

I have tried to balance speed and strength .

Algorithm :

The feedback function :

```
function cats_dogs(secret, guess):
   dogs = count of positions where secret[i] == guess[i]
   cats = count of letters overlapping between secret and guess - dogs
   return (cats, dogs)
```

—----------

Consistency check :

```
function is_consistent(candidate, guess, feedback):
   return cats_dogs(candidate, guess) == feedback
```

—----------

MinMax Strategy :

```
function next_guess_minimax(candidates):
    best_guess = None
    best_score = +∞
    for each g in candidates:
        partition_sizes = empty map
        for each c in candidates:
            fb = cats_dogs(c, g)
            partition_sizes[fb] += 1
        worst_partition = max(partition_sizes.values)
        if worst_partition < best_score:
            best_score = worst_partition
            best_guess = g
    return best_guess
```

Idea: pick the guess that minimizes the largest possible remaining set.

Heuristic Strategy :

```
function likely_fixed_positions(candidates):
    fixed = {}
    for position i in word length:
        if all candidates share same letter at i:
            fixed[i] = that letter
    return fixed

function next_guess_overlap(candidates):
    letter_counts = frequency of letters across all candidates
    fixed = likely_fixed_positions(candidates)

    for each word in candidates:
        score(word) = sum(letter_counts[unique letters in word])
        for each fixed position (i, ch):
            if word[i] == ch:
                score(word) += bonus
    return word with maximum score
```

Idea: bias guesses toward frequent letters and confirmed fixed positions.

Strategy controller :

```
function hybrid_next_guess(candidates, step, minimax_steps, switch_threshold):
    if step ≤ minimax_steps OR len(candidates) > switch_threshold:
        return next_guess_minimax(candidates)
    else:
        return next_guess_overlap(candidates)
```

Main solver loop

```
function solve_secret(words, secret):
    candidates = copy(words)
    step = 1
    history = []

    loop:
        guess = hybrid_next_guess(candidates, step, minimax_steps, switch_threshold)
        feedback = cats_dogs(secret, guess)
        history.append((step, guess, feedback, len(candidates)))

        if guess == secret:
            break

        candidates = [c for c in candidates if is_consistent(c, guess, feedback)]
        step += 1

    return history
```

Explanation of the algorithm

Feedback (cats, dogs):

Dogs = exact matches.

Cats = letters present but misplaced.

- Consistency check prunes words that don't match the observed feedback.
- Minimax strategy is expensive (quadratic over candidates) but very powerful in early game, ensuring worst-case partitions are small.
- Overlap heuristic is lightweight and exploits:

– Letter frequency (to test likely letters early).

– Fixed positions (to lock in “dogs”).

Hybrid control decides:

Early game (step ≤ minimax_steps OR candidate pool > threshold): use minimax to aggressively cut down the search space.

Late game: switch to overlap heuristic because candidate pool is small enough, and minimax is no longer worth the cost.

- Solver loop iteratively:

  > Makes a guess.

  > Records history.

  > Updates candidate pool.

  > Stops once the secret is found.

The toughest word for the algorithm was VANE with 15 guesses .
The average number of guesses was 6.58 .